Volume 11 Issue 03 Published, August 22, 2007 ISSN 1535-864X DOI: 10.1535/itj.1103.04

Intel® Technology Journal

Tera-scale Computing

Runtime Environment for Tera-scale Platforms

More information, including current and past issues of Intel Technology Journal, can be found at:
http://developer.intel.com/technology/itj/index.htm

Runtime Environment for Tera-scale Platforms

Bratin Saha, Programming Systems Lab, CTG
Ali-Reza Adl-Tabatabai, Programming Systems Lab, CTG
Richard L. Hudson, Programming Systems Lab, CTG
Vijay Menon, Programming Systems Lab, CTG
Tatiana Shpeisman, Programming Systems Lab, CTG
Mohan Rajagopalan, Programming Systems Lab, CTG
Anwar Ghuloum, Programming Systems Lab, CTG
Eric Sprangle, Visual Computing Group, DEG
Anwar Rohillah, Visual Computing Group, DEG
Doug Carmean, Visual Computing Group, DEG

Index words: runtime environment, scalable user level services, tera-scale platform

ABSTRACT

This paper presents the design and implementation of a runtime environment for tera-scale platforms. System software stacks currently view tera-scale platforms as an "SMP (symmetric multiprocessor) on a die." We show that there are fundamental differences between tera-scale and SMP systems that require that the software (SW) stack be re-architected. In particular, the SW stack needs to provide (1) support for efficient fine-grain parallelism, (2) programmability enhancements such as transactional memory, and (3) support for heterogeneous platforms and applications.

We discuss the design and implementation of a Many-Core RunTime (McRT) environment—a prototype tera-scale runtime environment—and show how it addresses the challenges of a tera-scale runtime. We also present simulation results from a tera-scale simulator to show that McRT enables excellent scalability on tera-scale platforms.

INTRODUCTION

System software tends to view a tera-scale chip multiprocessor (hereafter called TS-CMP) as a large-scale "symmetric multiprocessor (SMP) on a die"; yet, tera-scale CMPs have several characteristics that are fundamentally different from those of SMPs. It is critical to address these differences in order to implement a scalable and effective software stack. In particular it is important for the software stack to support (1) efficient fine-grain parallelism, (2) new concurrency abstractions that make parallel programming easier, and (3) platform and application heterogeneity.

Supporting Fine-grain Parallelism

TS-CMP has a very different compute-to-cache ratio than a traditional SMP. A 32-way SMP system typically has more than 100 MBs of aggregate cache size, while a 32-core TS-CMP has less than 10 MBs of cache. Thus a TS-CMP application needs to be threaded at a much finer granularity to reduce its working set. For example, MPEG4 encoding could be parallelized on a large-way SMP by encoding several frames in parallel. On a TS-CMP the encoding of an individual frame needs to be parallelized since the platform will not be able to cache multiple high-definition frames. Finally, many tera-scale applications benefit from fine-grain nested data parallelism rather than from coarse-grain task parallelism.

On the other hand, a TS-CMP enables fine-grain parallelism since inter-core communication is much easier—core-core bandwidth is of the order of terabytes/sec as opposed to gigabytes/sec for an SMP, and core-core latency is in the low tens of cycles (say 20 cycles) as opposed to hundreds of cycles in an SMP. Moreover, the effective core-core latency is much smaller, since the high degree of threading in a TS-CMP core allows some other thread (within the same core) to fully utilize the core resources if one thread is blocked on a cache miss.

Supporting New Concurrency Abstractions

Due to their high cost, large-way SMP systems have been restricted to niche markets, running applications written by sophisticated programmers whereas TS-CMP processors are targeted at mainstream price points and will bring parallelism to the average programmer. The success of TS-CMP processors and the applications that run on them depends on mainstream programmers embracing parallelism aggressively. Thus, the system SW stack should include new higher-level concurrency abstractions that make it easier for the average programmer to deal with parallelism.

Supporting Heterogeneity

Unlike SMPs, a TS-CMP software stack must comprehend heterogeneity at multiple levels. At the application level, TS-CMP processors will run a more diverse set of applications because they are targeted at a much broader market. At the hardware level, the TS-CMP platform may be heterogeneous with a combination of high-performance scalar cores, an array of high-throughput cores, and fixed function units. The system software stack must comprehend this heterogeneity. It needs to support configurable policies, for example, configurable scheduling policies, to adapt to different applications, and it needs to schedule applications according to their hardware requirements.

In this paper we present the design and implementation of McRT, a runtime environment for tera-scale platforms. McRT provides a configurable runtime framework that addresses the key tera-scale runtime requirements in the following ways:

- Fine-grain parallelism: McRT implements a significant fraction of threading services such as thread creation, synchronization, memory management, etc. at the user level. It also provides efficient user-level abstractions such as futures that make it easier to program and extract fine-grain parallelism.
- Concurrency abstractions: McRT includes a high-performance transactional memory library that supports an *atomic* construct in both C/C++ and Java. Transactional memory [15] provides a number of software engineering benefits compared to locks for managing access to shared data.
- Heterogeneity: McRT supports a number of configurable runtime policies that can be adapted for a particular application. In addition, McRT also supports multiple scheduling domains. Different hardware (HW) units can be mapped to different scheduling domains, and applications can be scheduled independently within each domain.

We show McRT's scalability using media encoding and Recognition, Mining, Synthesis (RMS) applications [11] on a tera-scale simulator. The results show that McRT's efficient threading primitives enable the applications to scale almost linearly up to 64 HW threads. We show that transactional memory can significantly ease parallel programming. Applications can use coarse-grain atomic blocks to synchronize access to shared data; yet they can achieve the performance of fine-grain locking. We also show a prototype implementation of a heterogeneous HW platform that leverages the support for scheduling domains in McRT.

McRT ARCHITECTURE

At its core, McRT contains a set of user-level threading primitives including a scheduler, memory manager, synchronization primitives, and a set of threading abstractions. We implemented these traditional operating system (OS) services as user-level primitives to improve efficiency by avoiding the expensive transitions between the user level and OS level making fine-grain parallelism more tractable. The McRT architecture is shown in Figure 1.

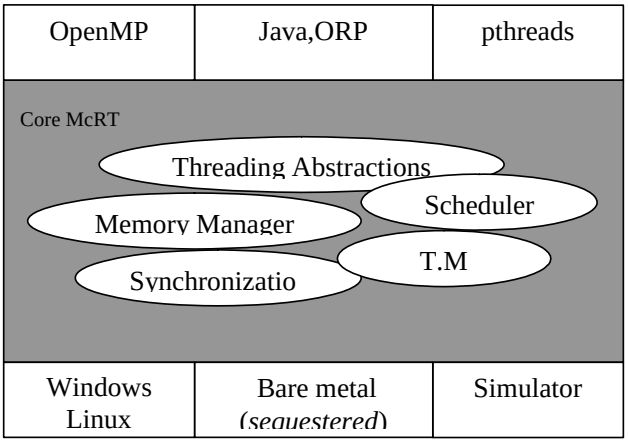

Figure 1: McRT architecture

McRT provides two user-level threading abstractions, threads and futures. The threads are similar to POSIX threads in functionality, while the futures are more lightweight and intended to support a concurrency idiom found in some languages such as MultiLisp [14] and CILK [8]. Futures provide a serial execution semantic, but can be executed in parallel if there are additional hardware resources.

The user-level scheduler is implemented as a task queue. An application can configure the number of task queues, e.g., specifying a single task queue for each processor. The application can also specify the scheduling policy, e.g., it can ask for a work-sharing policy where new tasks

are distributed among the task queues, or a work-stealing policy where idle processors search different queues for the next available task.

McRT uses a cooperative scheduling policy as opposed to the preemptive scheduling policy used predominantly in software stacks for SMPs. In an SMP system, the processing resource is expensive. Therefore the system software tries to timeshare the processing resource across multiple application threads by using preemptive scheduling. In a TS-CMP platform (say a platform with 128 cores), the processing resource is both inexpensive and abundant, which led us to use cooperative scheduling. This in turn addresses scalability bottlenecks, such as convoying, since an application can control when a thread gets preempted.

McRT includes a user-level synchronization library that includes different scalable algorithms such as MCS [24] locks and CLH [22] queues. It also includes a user-level memory allocator [16] that uses per-thread private allocation blocks. The allocator uses a completely non-blocking implementation that allows it to scale even with large oversubscription where the number of software threads is much greater than the number of hardware processors.

Finally, McRT includes a number of client adaptors that translate existing popular paradigms such as OpenMP and pthreads to the core McRT API. The OpenMP adaptor implements the API used by the Intel® C compiler, while the pthreads adaptor translates the POSIX API.

The core services in McRT are modularized and can be used as standalone services. For example, the memory manager ships as part of the Threading Building Blocks, while the transactional memory module has been tightly integrated into several compilers including the Intel C compiler, the StarJIT compiler [1], and the Harmony JITtrino compiler [5].

Evaluating Support for Fine-Grain Parallelism

We used a number of micro-benchmarks to evaluate the efficiency of the McRT threading primitives and hence its support for fine-grain parallelism. Figure 2 shows the results: the first row compares the cost of creating 255 threads; the second row compares the cost of 1000 consecutive lock acquire and release operations; and the final row compares the cost of 1000 context switches. In each case the *gettimeofday()* system call was used for the measurements. All the experiments were run on a 2.8GHz Intel® Xeon® processor. Column 2 reports the measurements observed by using native threads on Linux* 2.4.9, while Column 3 reports the measurements from using native threads on RedHat Enterprise* Linux 2.6.9-22ELsmp (NPTL 0.60).

	Native threads on Linux 2.4.9 µsec	Native threads on Linux 2.6.9 µsec	McRT µsec
Thread create (255 iterations)	21294	8960	1841
Mutex lock/release (1000 iterations)	120	82	81
Context switch (1000 iterations)	2927	3600	748

Figure 2: Micro-benchmark evaluation

We also measured the scalability of our threading primitives. Figure 3 compares the cost of creating thousands of threads on McRT and on Linux (2.6.9). Note that the efficiency of thread creation in McRT does not degrade even with thousands of threads.

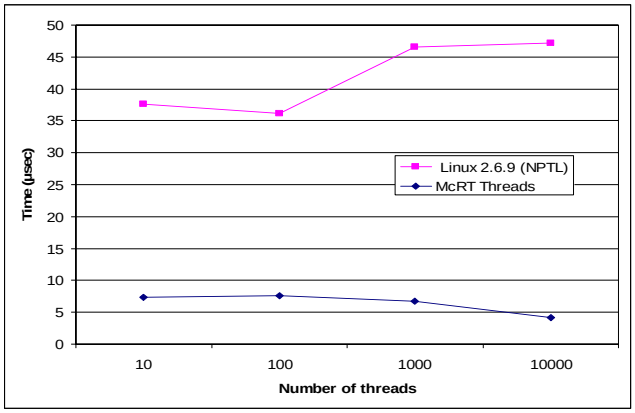

Figure 3: Scalability of thread creation

As mentioned before, McRT also implements futures to provide a lighter weight concurrency mechanism. Figure 4 compares the overhead of McRT futures to that of using McRT threads. For this, we created batches of futures and threads whose executable code simply returned immediately. We compared the time to complete such a batch using both threads and futures. Figure 4 compares the ratio of the execution time for threads and futures with futures being 40 to 100 times more efficient than threads. Obviously, futures can provide very good support for fine-grain parallelism.

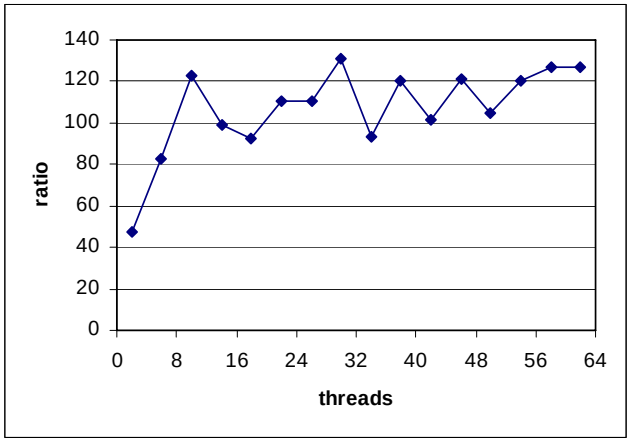

Figure 4: Thread vs. future creation overhead

NEW CONCURRENCY ABSTRACTION: TRANSACTIONAL MEMORY

Parallel programming poses many new challenges to the developer. A major one is synchronizing access to shared data between multiple threads. Traditionally, programmers have used locks for synchronization, but this method has well-known pitfalls such as deadlock and lack of composition. Transactional memory provides a new language construct that avoids the pitfalls of lock-based synchronization and eases concurrency control.

The McRT core services include a Transactional Memory (TM) library that supports the implementation of TM for C/C++ (unmanaged) and Java (managed). The TM library uses 2-phase locking to implement pessimistic write concurrency, and it uses versioning to implement optimistic read concurrency [27]. Every datum that may be accessed inside a transaction is associated with a transaction record—a pointer-sized word that mediates access to the shared datum. On a write, the TM library acquires exclusive ownership of the transaction record, performs an in-place update, and records the old value and the version number in an internal undo-log. On a read, the TM library records the version number of the transaction record (corresponding to the data address) in an internal read log. Before committing, the TM library validates a transaction by checking that the version numbers of the transaction records in the read set have not been changed. Upon committing, the lock is released and the version number is incremented. On an abort, the library uses the values in the undo-log to roll back the updates.

The TM library is also integrated with other runtime services such as memory management [16]. For example if a transaction allocates memory during its execution, the memory is automatically freed when the transaction aborts. Using a language-neutral API, we integrated the TM library with the Intel C/C++ compiler v10 and the StarJIT and JITtrino compilers for Java. These compilers take language-level *transactional* code blocks and insert calls to the appropriate runtime functions for every shared memory access inside the code block. This allows programmers to directly use TM rather than locks for concurrency control.

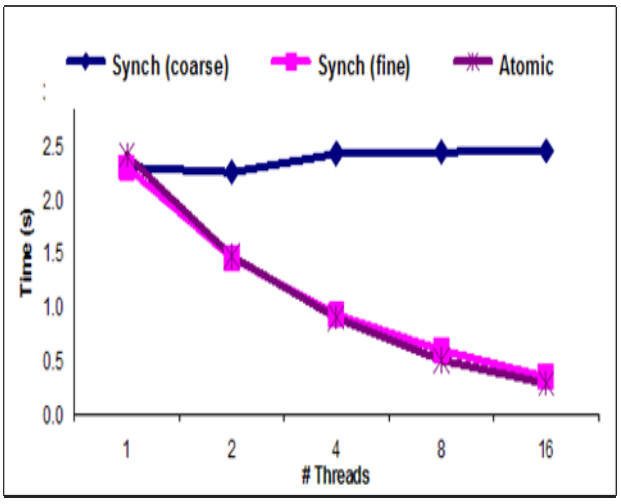

Figure 5: Transactions vs. locks on a hashmap

Figure 5 compares the performance of locks and transactions on a hashmap data structure. It measures the time taken to complete a set of insert, delete, and update operations on lock-based and transactional versions of the hashmap [2] on a 16-way SMP machine. The transactional version of the hashmap was obtained by replacing the critical sections in the coarse-grain synchronization version with atomic sections. As expected, coarse-grained locking (Sync(coarse)) does not scale, but both the transactional and the fine-grain version scale comparably, even though the transactional version uses coarse-grain synchronization.

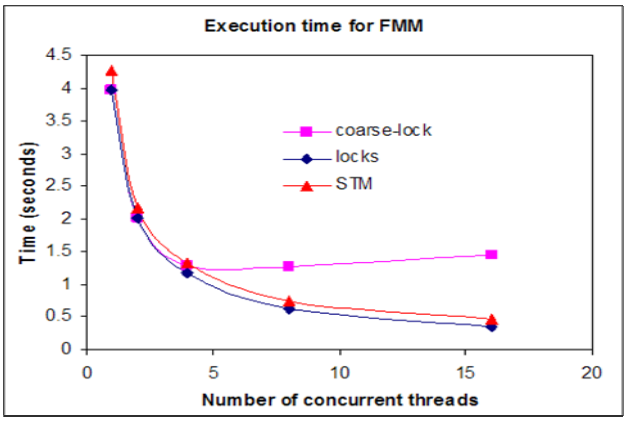

Figure 6: Transactions vs. locks on FMM (SPLASH2)

Figure 6 shows a similar result on the FMM benchmark, which is part of the SPLASH2 suite. The transactional version of the benchmark was obtained by replacing the coarse-grain critical sections with transactions. Again, the transactional version scales as well as the fine-grain version even though it uses coarse-grain synchronization.

These results show that McRT allows a programmer to leverage the software engineering benefits of transactional memory, such as composability and coarse-grain reasoning, while getting the performance of fine-grain locking. We expect this to be a key enabler in promoting multithreaded programming for tera-scale platforms.

SUPPORTING HETEROGENEITY

McRT uses multiple scheduling domains to support platform heterogeneity. Each domain can represent a set of hardware units with specific features such as good scalar performance, good throughput, special instruction sets, and so on. McRT extends the task queue mechanism to support scheduling domains. To create a domain D_k consisting of logical processors P_i to P_j a client creates a task queue Q_k that is accessed only by the processors P_i to P_j. New tasks created at these processors are only added to Q_k. The scheduler also exports an API that allows a task to yield its current logical processor and enqueue itself on a different task queue. A task executing in a domain D_k can switch to a different domain D'_k by enqueing itself on to the task queue Q'_k at which point it will get executed by the processors in D'_k . Applications, or even different parts of the same application, can be scheduled on different hardware units based on their requirements.

We prototyped a heterogeneous hardware platform on an 8-way SMP system. One processor (referred to as the OS processor) in the system boots up Windows* Server 2003, while the remaining seven processors (referred to as the sequestered processors) use McRT for all the threading services, without using the OS. The sequestered processors use a lightweight executive for interrupt handling. Thus, the sequestered processors emulate an attached compute engine, with the OS processor emulating a host CPU. Internally, McRT creates two scheduling domains, one representing the sequestered processors and the other representing the OS processor. The system configuration is shown in Figure 7.

We ran Equake* using the standard Spec input on the sequestered system. At the beginning the application is serial and reads the input. It then forks off a number of threads to perform the computation. McRT scheduled the serial part on the OS core and the parallel part on the sequestered cores.

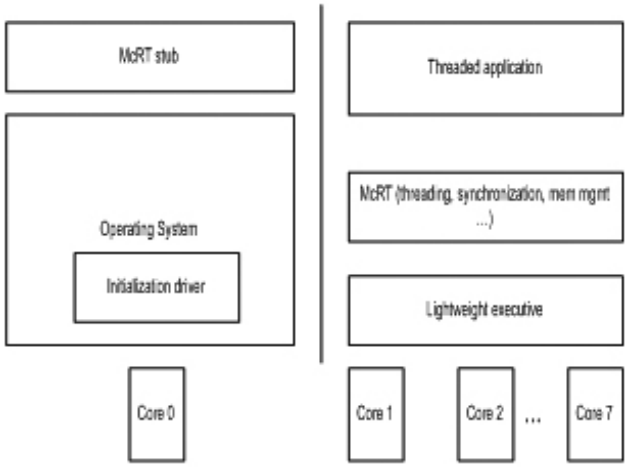

Figure 7: Sequestered system

Figure 8 shows the performance of Equake on the sequestered system. We first ran the benchmark on the 8-way SMP system with Windows running on all the processors. These numbers are reported as "Native" and "McRT-OS": "Native" refers to the performance from the Intel OpenMP* implementation (referred to as KAI in the figure), and "McRT-OS" refers to the performance from running McRT on top of Windows on the 8-way SMP. "McRT-Sequestered" refers to the performance on the sequestered system with one OS processor and seven sequestered processors. All speedups are reported with respect to the single thread "Native" execution time.

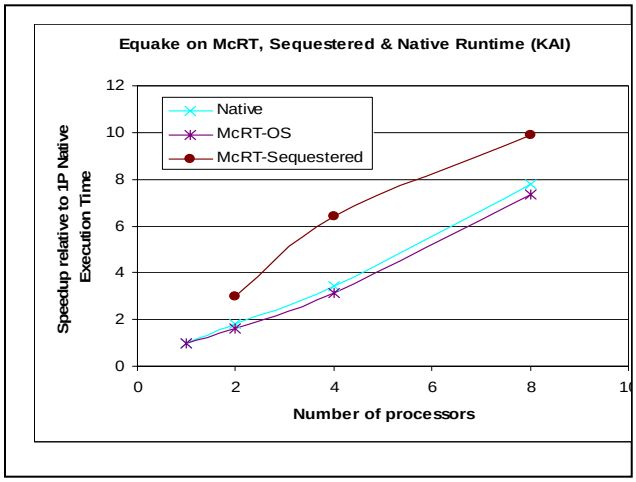

Figure 8: Equake on sequestered system

Equake performs much better on the sequestered system, mainly due to the fact that the software stack is much more lightweight and is not interrupted as often. Another reason is that in the sequestered mode, the application reserves and locks down enough memory at initialization so that it does not encounter page faults during execution.

RESULTS

We used a cycle-accurate simulator to evaluate McRT's performance on a TS-CMP processor. The simulated platform consists of an array of up to 16 in-order cores, each of which has four threads. Each core will select a different thread each cycle, round-robin, unless the thread is stalled due to, for example, a cache miss, being in the sleep state. The memory system consists of a 32 KB L1 data cache that is shared by all four threads in the core, a 2MB L2 cache that is shared by all the cores, and an off-chip 4MB L3 cache. All caches were simulated with an 8-way set associative configuration. The L1 cache access time is 3 cycles, the L2 cache access time is 12 cycles, and the L3 cache access time is 40 cycles. The simulator performs a cycle accurate simulation of the execution pipeline for all the HW threads, the different caches, the coherence protocol, the bandwidth for data transfer between different parts of the memory system, and the interconnect to the external memory.

We ported McRT to run directly on the simulator. Thus, the results reflect true execution driven simulation and accurately account for inter-thread synchronization. The simulator was modified to support system calls, while McRT provided all the threading services required by the application.

We used the popular open source MPEG4 encoder XviD (www.xvid.org*) and a set of RMS kernels [11] for Singular Value Decomposition (SVD) and Self Organizing Maps (SOM) as our workloads. The XviD encoder is used mainly on frames of 1920x1080 to correspond with frame sizes in emerging high-definition video. We show the performance for encoding the P frames since these (along with the B frames) happen to be the computationally intensive parts of the encoding. The simulated cache size does not allow multiple frames to be encoded in parallel; therefore, we had to parallelize the encoding of a single frame. A frame is partitioned into "k" sub-blocks, where "k" is the number of logical processors used for encoding. Thus, the scalability of MPEG4 encoding is a good test of the efficiency of McRT's fine-grain threading support.

SVD has numerous applications in the areas of data-mining and feature extraction, signal processing, and automated control; this workload uses the Jacobi method. An SOM is an unsupervised learning method represented by a two-layer neural network. Typically, it is used to map N dimensional data to two dimensions to discern patterns. It is extensively applied in text and feature mining, pattern recognition, and medical diagnostics.

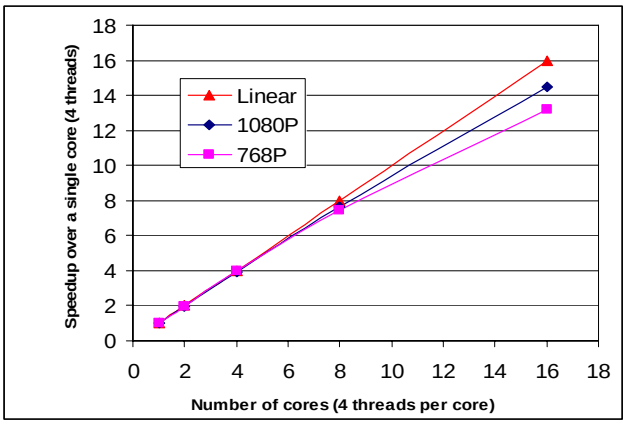

Figure 9: XviD speedup

Figure 9 shows the speedup of encoding a single frame as we increase the number of processors. The *x-axis* shows the number of cores. Note that for k cores, the number of HW processors is k*4. The graph uses the execution time on a single core (4 threads) as the baseline. Even at 16 cores (64 threads) we get almost a linear speedup (the "Linear" line in the graph represents speedup expected if the application was completely parallelized). The speedup on the 1080P (1920x1080) frame is slightly higher than on the 768P (1024x768) frame since the sub-block sizes are larger, and hence the cost of threading gets amortized.

Figure 10 shows the speedup for the RMS workloads. (The x-axis represents the number of cores, and the baseline is the execution time on a single core.) Both SVD and SOM scale almost perfectly up to 64 HW threads.

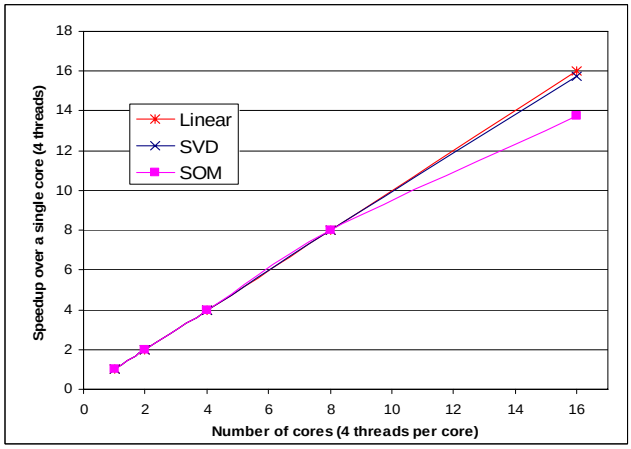

Figure 10: RMS speedup

RELATED WORK

Our work was inspired by previous projects in the areas of language systems, operating systems design, and high-performance computing. Several operating systems

have explored issues related to SMP scalability [3, 4, 17, 23, 26]. Disco [9] and K42 [18] have explored the design of scalable operating systems. Currently, operating systems such as Linux and MS Windows treat CMP architectures as an SMP on a chip. While the McRT framework arose out of our experience of these projects, this paper shows why we still need to rethink the system design to fully enable a scalable tera-scale environment.

The threading and synchronization primitives provided by McRT are similar to those seen in traditional user-level threading packages such as pthreads [19], NGPT [25], NPTL [10] and Capriccio [6]. The McRT scheduler is comparable to user-level schedulers that have been explored in the context of kernels such as L4 [20], Exokernel [12], Flux [13], and SPIN [7], but it is more lightweight and configurable. The McRT runtime is also comparable to language runtimes such as CILK [8] and OpenMP [21], but unlike these runtimes it is platform-neutral.

Finally, the specific mechanisms used in the McRT-STM [27] and the McRT memory manager [16] are described elsewhere, while the compiler integration is discussed in [2, 28].

CONCLUSION

A tera-scale platform has a number of fundamental differences from a traditional SMP system, which requires that the system software stack be redesigned to provide an effective and scalable runtime environment. In particular, the runtime environment must provide good support for fine-grain parallelism, support new concurrency abstractions that ease parallel programming, and support heterogeneous platforms and applications.

This paper described how McRT addresses the challenges of a many-core environment. To enable efficient fine-grain parallelism, McRT replaces many of the OS-level services with user-level primitives. Our results show that this enables a very scalable runtime stack that scales to more than 64 HW threads.

To ease parallel programming, McRT provides a high-performance TM library that supports a language-level *atomic* construct. TM provides several software engineering benefits compared to locks such as deadlock freedom, scalable composition, and failure atomicity. Additionally, our results show that transactions achieve the performance of fine-grain locking, yet allow coarse-grain synchronization.

McRT supports heterogeneity by dividing the platform into independent scheduling domains. These domains can be mapped to different hardware resources, and applications can be scheduled on the domain that best fits their requirements. In addition, McRT also supports a number of configurable runtime policies that allow it to adapt to different applications.

ACKNOWLEDGMENTS

We thank Leaf Petersen who provided the futures implementation in McRT and Cheng Wang who helped to implement the transactional memory support in the Intel C compiler. We also thank Jesse Fang who provided unstinted support and guidance for this work.

REFERENCES

[1] A. Adl-Tabatabai, J. Bharadwaj , D. Chen, A. Ghuloum, V. S, Menon, B. R. Murphy, M. Serrano, T. Shpeisman, "The StarJIT compiler: a dynamic compiler for managed runtime environments," *Intel Technology Journal*, Feb. 2003.

[2] A. Adl-Tabatabai, B.T. Lewis, V.S. Menon, B.M. Murphy, B. Saha, T. Shpeisman, "Compiler and runtime support for efficient software transactional memory," *PLDI*, 2006.

[3] T. E. Anderson, D. E. Lazowska, and H. M. Levy, "The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors," *IEEE Trans. on Comp*., Dec. 1989.

[4] T.E. Anderson, B.N. Bershad, E.D. Lazowska, and H.M. Levy, "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism.,"*ACM ToCS*, Feb. 1992

[5] Apache Harmony Project at http://harmony.apache.org/*

[6] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer, "Capriccio: Scalable threads for internet services," in *Proceedings SOSP-19*, 2003.

[7] B. N. Bershad, S. Savage, P. Pardyak, E. Gün Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, S. J. Eggers, "Extensibility, Safety and Performance in the SPIN Operating System," *SOSP*, 1995.

[8] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk, "An Efficient Multithreaded Runtime System," *PPoPP*, 1995.

[9] E. Bugnion, S. Devine, and M. Rosenblum, "Disco: running commodity operating systems on scalable multiprocessors," In *Proceedings SOSP-16*, 1997.

[10] U. Drepper and I. Molnar, "The native POSIX thread library for Linux, "January 2003, at http://people.redhat.com/drepper/nptl-design.pdf*.

[11] P. Dubey, "Recognition, Mining, and Synthesis moves computers to the era of tera," *Technology@Intel*, February 2005.

[12] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr., "Exokernel: an operating system architecture for application-specific resource management," *SOSP-15*, 1995.

[13] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers, "The Flux OSKit: A Substrate for Kernel and Language Research," *SOSP-16*, 1997.

[14] R. Halstead, "Multilisp: A Language for Concurrent Symbolic Computation," in *ACM Transactions on Programming Languages and Systems*, October 1985.

[15] M. Herlihy, and J. E. B. Moss, "Transactional memory: architectural support for lock-free data structures," *ISCA*, 1993.

[16] R. Hudson, B. Saha, A. Adl-Tabatabai, B. Hertzberg, "McRT-Malloc: A Scalable Transaction Aware Memory Allocator," *ISMM*, 2006.

[17] M. B. Jones, R. F. Rashid, "Mach and Matchmaker: Kernel and Language Support for Object-Oriented Distributed Systems," *OOPSLA*, 1986.

[18] The K42 project, IBM Research, at http://www.research.ibm.com/k42/*

[19] B. Lewis and D. J. Berg, *Multithreaded Programming with Pthreads*, Prentice Hall, New Jersey, 1998.

[20] J. Liedtke, "On micro-Kernel Construction," *SOSP-15*, 1995.

[21] H. Lu, Y. C. Hu, and W. Zwaenepoel, "OpenMP on networks of workstations," in *Supercomputing*, November 1998.

[22] P. Magnussen, A. Landin, and E. Hagersten, "Queue locks on cache coherent multiprocessors," *8th Intl. Parallel Processing Symposium*, Cancun, Mexico, April 1994.

[23] B. D. Marsh, M. L. Scott, T. J. LeBlanc, and E. P. Markatos, "Firstclass user-level threads," in *Proceedings SOSP-13*, October 1991.

[24] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

[25] Next Generation POSIX Threading at http://www-124.ibm.com/pthreads/*

[26] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch, "The Sprite network operating system," *IEEE Computer*, 21(2):23–36, February 1988.

[27] B. Saha, A. Adl-Tabatabai, R. Hudson, C. Minh, B. Hertzberg, "McRT-STM: A High Performance Software Transactional Memory System For A Multi-Core Runtime," *PPoPP*, 2006.

[28] C. Wang, W. Chen, Y. Wu, B. Saha, A. Adl-Tabatabai, "Code Generation and Optimization for Transactional Memory Constructs in an Unmanaged Language," *CGO*, 2007.

AUTHORS' BIOGRAPHIES

Bratin Saha is a Senior Staff Researcher in Intel's Programming Systems Lab. His current research is focused on the design and implementation of modern concurrency abstractions, such as transactional memory, and highly concurrent runtime environments. He was one of the architects of locking and synchronization in the Nehalem processor. Bratin received his M.S. and Ph.D degrees in Computer Science from Yale University, and his B.S. degree in Computer Science and Engineering from the Indian Institute of Technology, Kharagpur. His e-mail is bratin.saha at intel.com.

Ali-Reza Adl-Tabatabai is a Senior Principal Engineer in Intel's Programming Systems Lab. He leads a team of researchers working on compilers and scalable runtimes for future Intel® Architectures. Ali has spent most of his career building high-performance programming language implementations, including static and dynamic optimizing compilers and language runtime systems. His current research concentrates on language features, that make it easier for the mainstream developer to build reliable and scalable parallel programs for future multi-core architectures, and on architectural support for those features. Ali has published over 20 papers in leading conferences and journals. He received his Ph.D. degree in Computer Science from Carnegie Mellon University.

Richard L. Hudson is best known for his work in memory management including the invention of both the Train Algorithm and the Sapphire Algorithm. Richard joined Intel in 1998 where he has worked on concurrency related issues. He went to Shortridge and holds a B.A. degree from Hampshire College and an M.S. degree from the University of Massachusetts. His e-mail is rick.hudson at intel.com.

Vijay Menon is a Senior Research Scientist in the Programming Systems Lab at Intel investigating new programming technologies for multi-core systems. His primary areas of a research include compilers, managed runtimes, and transactional memory. Vijay holds a Ph.D.

degree in Computer Science from Cornell University and a B.S. degree in Electrical Engineering and Computer Science from the University of California at Berkeley.

Tatiana Shpeisman received her B.S. degree in Applied Mathematics from Leningrad Electrical Engineering Institute. She got her M.S. and Ph.D. degrees in Computer Science from the University of Maryland, College Park. Currently, she is a Senior Software Engineer working at the Intel Microprocessor Technology Lab. She is a member of ACM. Her e-mail is Tatiana.shpeisman at intel.com.

Mohan Rajagopalan's current research focuses on parallel runtime technologies for emerging many-core platforms. Mohan received his M.S. and Ph.D degrees from the University of Arizona in 2001 and 2006, respectively. His dissertation explored the application of language and compiler techniques to optimize overall systems design for automatically improving aspects such as security and dependability in addition to performance. His e-mail is mohan.rajagopalan at intel.com.

Anwar Ghuloum is a Principal Engineer with Intel's Microprocessor Technology Lab, working on diverse topics such as parallel language and compiler design, parallel architecture evaluation, optimizing memory system performance, and multimedia applications. Anwar received a B.S. degree in Computer Science and Engineering from the University of California, Los Angeles and a Ph.D. degree in Computer Science from Carnegie Mellon University's School of Computer Science in 1996. Before joining Intel, he co-founded and was the CTO of a fab-less semiconductor startup that designed parallel image and video processors for the consumer electronics market. Prior to that, Anwar developed novel predictive drug design software for early lead optimization using 3D surface pattern recognition techniques for a biotech startup. A recurring theme in Anwar's work has been to bridge high-level application knowledge and low-level parallel architecture constraints with careful parallel language and compiler design to achieve the optimal tradeoffs in productivity and performance. His e-mail is anwar.ghuloum at intel.com.

Eric Sprangle is a Principal Engineer with Intel's Visual Computing Group in Austin. Eric has been with Intel for eight years, working on the Intel® Pentium® 4 processor family, and he is currently one of the lead architects on the Larrabee project. Prior to joining Intel, Eric worked at ROSS Technology. Eric enjoys training for and racing in triathlons. His e-mail address is eric.sprangle at intel.com.

Anwar Rohillah is currently working as a VCG architect focusing on performance analysis and simulation. He has also worked on the Intel Pentium 4 processor family developing hardware prefetchers and doing performance analysis. Anwar obtained his B.A.Sc. degree in Computer Engineering from the University of Waterloo and joined Intel in 1999. His e-mail is anwar.rohillah at intel.com

Doug Carmean is a Senior Principal Engineer with Intel's Visual Computing Group in Oregon. Doug was one of the key architects responsible for definition of the Intel Pentium 4 processor. He has been with Intel for 18 years, working on IA-32 processors from the 80486 to the Intel Pentium 4 processor and beyond. Doug is currently the Larrabee Chief Architect. Prior to joining Intel, Doug worked at ROSS Technology, Sun Microsystems, Cypress Semiconductor and Lattice Semiconductor. Doug enjoys fast cars and scary, Italian motorcycles. His e-mail address is douglas.m.carmean at intel.com.

THIS PAGE INTENTIONALLY LEFT BLANK

For further information visit:

developer.intel.com/technology/itj/index.htm